Scheduling Parallelizable Jobs Online to Minimize the Maximum Flow Time

Kunal Agrawal Jing Li Kefu Lu Benjamin Moseley
Washington University in St. Louis.
{kunal, li.jing, kefulu, bmoseley}@wustl.edu

ABSTRACT

In this paper we study the problem of scheduling a set of dynamic multithreaded jobs with the objective of minimizing the maximum latency experienced by any job. We assume that jobs arrive online and the scheduler has no information about the arrival rate, arrival time or work distribution of the jobs. The scheduling goal is to minimize the maximum amount of time between the arrival of a job and its completion — this goal is referred to in scheduling literature as *maximum flow time*. While theoretical online scheduling of parallel jobs has been studied extensively, most prior work has focussed on a highly stylized model of parallel jobs called the "speedup curves model." We model parallel jobs as *directed acyclic graphs*, which is a more realistic way to model dynamic multithreaded jobs.

In this context, we prove that a simple First-In-First-Out scheduler is $(1+\epsilon)$-speed $O(\frac{1}{\epsilon})$-competitive for any $\epsilon > 0$. We then develop a more practical work-stealing scheduler and show that it has a maximum flow time of $O(\frac{1}{\epsilon^2}\max\{\mathrm{OPT}, \ln(n)\})$ for n jobs, with $(1+\epsilon)$-speed. This result is essentially tight as we also provide a lower bound of $\Omega(\log(n))$ for work stealing. In addition, for the case where jobs have weights (typically representing priorities) and the objective is minimizing the maximum weighted flow time, we show a non-clairvoyant algorithm is $(1+\epsilon)$-speed $O(\frac{1}{\epsilon^2})$-competitive for any $\epsilon > 0$, which is essentially the best positive result that can be shown in the online setting for the weighted case due to strong lower bounds without resource augmentation.

After establishing theoretical results, we perform an empirical study of work-stealing. Our results indicate that, on both real world and synthetic workloads, work-stealing performs almost as well as an optimal scheduler.

1. INTRODUCTION

In today's systems, response time, or latency, is often a very important measure of performance. For interactive services on clouds and servers, the platform scheduler is often interested in minimizing the maximum latency experienced by a job once it has been submitted to be processed. In addition, these services often run on large parallel machines with many processors and it is important to utilize these parallel machines efficiently to process these requests. In this paper, we consider the problem of minimizing the maximum latency or ***maximum flow time***. Formally, given n parallel jobs and a parallel machine with m processors, the scheduling goal is to minimize the amount of time between a job's arrival and its completion, over all jobs. We consider the online version of this problem, where jobs arrive dynamically. The objective of maximum flow time is the natural generalization of the makespan[1] objective to the case where jobs arrive over time. We also assume that the scheduler has no knowledge of the job arrival times or work distribution.

This paper focuses on parallel programs expressed through *dynamic multithreading*[2] (see [9, Ch. 27]), which is common in many parallel languages and libraries, such as Cilk dialects [14, 19], Intel TBB [25], Microsoft Parallel Programming Library [7] and OpenMP [23]. In these parallel languages, programmers express algorithmic parallelism, through linguistic constructs such as "spawn" and "sync," "fork" and "join," or parallel for loops. On the other hand, programmers do not need to provide any mapping from subcomputations to processors — it is the job of the scheduler to execute the work of each job onto processors efficiently.

Scheduling a single dynamic multithreaded program has been studied extensively in the parallel computing literature. Parallel runtime systems generally use work-stealing as a scheduler since it is known to be an efficient scheduler for such programs both theoretically and in practice [14, 5]. A single parallel program having W *work* — the running time on 1 processor, and P *critical-path length* — the length of the critical path (the longest path in the program), can be executed in $O(W/m + P)$ (expected) time on m processors (or workers) using a work-stealing scheduler. This running time is asymptotically optimal and guarantees linear speedup for programs with sufficient parallelism.

However, the problem of how to schedule these parallel programs in multiprogrammed environments where a quality of service guarantee must be provided is not well studied. There has been some prior work on how to allocate processors to programs in a fair and efficient manner [1] and some further work on using it to provide mean completion time

SPAA '16, July 11-13, 2016, Pacific Grove, CA, USA

DOI: http://dx.doi.org/10.1145/2935764.2935782

[1]The makespan of a schedule is the time that last job completes and this objective is popular when all jobs arrive at the same time.

[2]Dynamic multithreading is also called fork-join parallelism.

guarantees [16] and average flow time guarantees [2], but none of the work considers maximum flow time.

On the other hand, the multiprogrammed scheduling problem is well-studied for the case where each job is sequential, i.e. can only use one processor at a time. In particular, it is known that the algorithm First-In-First-Out (FIFO) is $(3/2 - \frac{1}{m})$-competitive [3, 4]. A related problem that has been considered for sequential jobs is when jobs have weights where weight represents some sort of priority of the job (not necessarily correlated with the job's work). In this case, the scheduler is interested in minimizing the maximum weighted flow time. For this setting, it is known that any algorithm is $\Omega(W^{.4})$-competitive where W is the ratio of the maximum weight to minimum weight. This is true even when jobs are sequential and unit sized [8].

Due to this strong lower bound, previous work has considered a *resource augmentation* analysis [20] where the algorithm is given extra speed over the adversary. An ***s-speed c-competitive*** algorithm achieves a competitive ratio of c when given processors s times the speed of the optimal schedule. An ideal algorithm is $(1+\epsilon)$-speed $O(f(\epsilon))$-competitive for any $\epsilon > 0$ where $f(\epsilon)$ is some function that only depends on ϵ. That is, an algorithm which achieves constant competitiveness with the minimum possible resource augmentation. Such an algorithm is referred to as *scalable*. Finding a scalable algorithm is the best positive result one can hope for when strong lower bounds exists without resource augmentation. For jobs with weights, a scalable algorithm is known when jobs are sequential [8].

In this paper, we present several theoretical results for minimizing maximum flow time for dynamic multithreaded jobs. These are the first known non-trivial results for maximum flow time in the DAG model. We refer the reader to Section 8 for work in related parallel models. All of the algorithms considered in this paper are *non-clairvoyant*, meaning that they have no prior knowledge of the size or structure of the jobs or when they arrive. In particular, our contributions are as follows:

1. We start with an idealized FIFO scheduler — at each time step, FIFO looks at jobs in the order of arrival and allocates each job as many processors it can use until it runs out of jobs or processors. We prove that FIFO is $(1+\epsilon)$-speed $O(\frac{1}{\epsilon})$-competitive (Section 3).
2. We then generalize the result to work stealing (Section 4). Work stealing is a practical and efficient scheduler that is used in many parallel languages and libraries. In comparison, an implementation of the ideal FIFO scheduler is likely to have high overhead since it is centralized and potentially preempts jobs and re-allocates processors at every time step. For work stealing, we prove that a version of it, called *admit-first*, is scalable for "reasonable jobs". In particular, we show that admit-first with $(1+\epsilon)$-speed has maximum flow time $O(\frac{1}{\epsilon^2}\max\{\text{OPT}, \ln(n)\})$ over n jobs for any fixed $\epsilon > 0$ with high probability. Note that if any job has span $\Omega(\lg n)$ or work $\Omega(m \lg n)$, then $\text{OPT} \geq \ln n$ and admit-first is scalable with $(1+\epsilon)$-speed $O(\frac{1}{\epsilon^2})$-competitive with high probability.
3. We introduce a generalization of admit-first scheduler, called steal-k-first. Our goal in this generalization is to design a work-stealing scheduler that is closest to FIFO since intuitively FIFO is the 'right' scheduling policy for maximum flow time, but is inefficient in implementation. Steal-k-first is parameterized by k. Intuitively as k becomes larger, this algorithm becomes closer to the FIFO scheduler. Theoretically, this scheduler is $(k+1+\epsilon))$-speed $O(\frac{1}{\epsilon^2}\max\{\text{OPT}, \ln(n)\})$-competitive for any $\epsilon > 0$ and $k \geq 0$. It reduces to admit-first when $k = 0$.
4. We also provide a lower bound showing that the competitive ratio of work-stealing is $\Omega(\lg n)$ — that is, if all jobs are tiny with work $o(\lg n)$, then work stealing cannot be scalable due to the randomization involved (Section 5). This shows that our upper bound is close to being tight.
5. We implemented admit-first and steal-k-first in Thread Building Block (TBB) and compare their performance with a simulated optimal scheduler on realistic and synthetic workloads. Evaluation results shows that a work stealing scheduler (especially steal-k-first) have comparable performance to the optimal scheduler (Section 6).
6. Finally, we consider the case where jobs have weights and show a non-clairvoyant algorithm Biggest-Weight-First (BWF) is $(1+\epsilon)$-speed $O(\frac{1}{\epsilon^2})$-competitive for any $\epsilon > 0$, which is the best positive result that can be shown in the online setting for the weighted case (Section 7).

2. PRELIMINARIES

In the online scheduling problem of multiple jobs, n jobs arrive over time and are scheduled on m identical processors. Each job J_i has an arrival (release) time r_i, which is the first time an online scheduler is aware of the job. Each job could have a weight w_i — this weight is known to the scheduler when the job arrives and may not be correlated to the work of the job. For the unweighted setting, $w_i = 1$ for all jobs.

When analyzing the performance of a scheduling algorithm, we denote c_i as the completion time of job J_i in the algorithm's schedule. We denote $F_i = c_i - r_i$ as the flow time of job J_i in the algorithm's schedule. The goal of the scheduler is to minimize $\max_{i\in[n]} w_i F_i$.

A dynamic multithreaded job J_i can be represented as a ***Directed-Acyclic-Graph (DAG)*** G_i. Each node (task) v in G_i has an associated processing time p_v and the node must be processed sequentially on a processor for p_v time to be completed. A node in G_i cannot be executed until all of its predecessors in G_i have been executed. We say that a node is *ready* if all of its predecessors have been processed. Multiple ready nodes for the same job can be scheduled simultaneously. A job is completed only once all of the nodes in its DAG have been completely processed. Note that we do not assume that the scheduler knows the DAG in advance; in fact, the DAG unfolds dynamically as the job executes.

Symbol	Definition
c_i	completion time of job J_i in schedule
r_i	arrival time of job J_i
F_i	flowtime of job J_i
P_i	the critical path length of J_i
m	the number of processors
w_i	the weight of job J_i
OPT	optimal schedule and also optimal objective

Table 1: Symbols and Definitions

Dynamic multithreaded jobs can be characterized by two important parameters. The ***critical-path length*** P_i of job J_i is defined to be the execution time of J_i if it were scheduled continuously on an infinite number of processors. Alternatively, it is defined to be the length of the longest path

in G_i, where each node v in the longest path contributes p_v to the length of the path. Note that P_i is a lower bound on the execution time of J_i for any scheduler. The ***Work*** W_i of job J_i is the execution time on 1 processor; or alternatively, the sum of the processing times of all the nodes in the DAG. We summarize some of the notations in Table 1.

The following proposition is well-known and has been shown in several works, for example [22]. The proposition states that any time a scheduler is working on all the ready nodes for some job J_j, then the scheduler must be decreasing the remaining critical path of J_j.

PROPOSITION 2.1. *If during each time step during a time interval $[t', t]$, a scheduler of speed s is always scheduling all available nodes for a job J_j, then the scheduler reduces the critical path length of J_i by $s(t - t')$.*

Difficulties of Analyzing Algorithms in the DAG Model. For scheduling sequential jobs, previous analyses for maximum flow time follows by showing that the algorithm A under consideration cannot fall behind for more than mp_{max} where p_{max} is the maximum processing time of a job. The is because either the algorithm A has less than m jobs and so there can be at most mp_{max} total work for unsatisfied jobs in its queue; or it has more than m jobs, but then the algorithm A cannot fall behind much since it will always be using all m processors.

Unfortunately, this argument is no longer straightforward for DAGs. To see this, consider unweighted flow time and let OPT denote the objective of the optimal solution. Note that in the sequential setting, $\text{OPT} \geq p_{max}$. In contrast, in the DAG model, some jobs could have work $\Theta(m\text{OPT})$. Now, even if the algorithm A under consideration has fewer than m unsatisfied jobs, it can have a total work of $\Theta(m^2\text{OPT})$ in its queue — allowing it to fall very far behind OPT. Thus, it is difficult to directly bound the amount of work an algorithm falls behind the OPT just as a function of the number of jobs in the algorithm's schedule. To prove that the algorithm does not fall far behind the optimal solution, a necessary condition for an algorithm to be competitive for maximum flow time, we instead identify times where the algorithm must not be too far behind the optimal solution and then show that the algorithm must not fall behind much further following those times.

On an additional note, most previous work on scheduling parallel jobs is in the arbitrary speedup curves model (see Section 8) for some details) and uses a potential function argument (see [17] for a tutorial). Unfortunately, there are currently no known potential function proofs for maximum flow time unlike for other objectives.

3. UNWEIGHTED MAXIMUM FLOW TIME USING FIFO

In this section our goal is to prove the following theorem stating that the algorithm First-In-First-Out (FIFO) is $(1+\epsilon)$-speed $O(\frac{1}{\epsilon})$-competitive for minimizing the maximum unweighted flow time for any $0 < \epsilon < 1$.

THEOREM 3.1. *First-in-First-Out (FIFO) is $(1+\epsilon)$ speed $O(\frac{1}{\epsilon})$ competitive for minimizing the maximum unweighted flow time for any $\epsilon > 0$.*

FIFO is defined as follows. At any time t, FIFO orders the jobs in increasing order by their arrival time, breaking ties arbitrarily. The algorithm then assigns all of the ready nodes for the first job to unique processors, then recursively does the same for the next job in the list. This continues until all processors have been assigned some node or there are no more ready nodes available. The algorithm may have a choice on which ready nodes of a job to schedule if the job has more ready nodes than the number of processors that have not been assigned to a node when the job is considered. In this case, we assume the scheduler chooses an arbitrary set of ready nodes from the job.

The rest of this section is devoted to proving Theorem 3.1. We assume for the remaining of this section that FIFO is given $(1+\epsilon)$-speed for some constant $0 < \epsilon < 1$ and we will show that FIFO is $\frac{3}{\epsilon}$ competitive. To show this, assume for the sake of contradiction that FIFO is more than $\frac{3}{\epsilon}$-competitive and we consider the instance for which FIFO does not achieve a competitive ratio of $\frac{3}{\epsilon}$. Let J_i be the job with the maximum flow F_i at this instance, so $\text{OPT} < \frac{\epsilon}{3}F_i$ by assumption. Since no jobs that arrive later than J_i has any effect on how or when J_i is scheduled due to FIFO's scheduling policy, in our instance J_i is the last job to arrive.

We begin by showing that during the time interval job J_i is alive in FIFO's schedule, the processors must be busy for most of the interval. We define one ***time step*** as the time period for a s-speed processor to execute one unit of work. In other words, in one time step m processors with speed s can finish m work of jobs. Note that on processors with different speeds, the length of a time step will be different. Hence, the number of time steps on a s-speed processor in T time is sT, while it is T on unit speed processor.

LEMMA 3.2. *During the interval $[r_i, c_i]$ in FIFO's schedule, there can be no more than $\frac{\epsilon}{3}F_i$ time steps where not all m processors are busy working on jobs.*

PROOF. For the sake of contradiction, suppose there is at least $\frac{\epsilon}{3}F_i$ time steps during $[r_i, c_i]$ where not all processors are busy. Consider FIFO's scheduling policy. Anytime during $[r_i, c_i]$ where FIFO is not processing nodes on every processor, FIFO must be scheduling all of the ready nodes of J_i. Due to this, at these times FIFO is working on the critical path length of J_i by Proposition 2.1. Let this path length be P_i, then we have $P_i \geq \frac{\epsilon}{3}F_i$.

Also note that OPT cannot finish a job in less time than its critical-path length, this leads to $\text{OPT} \geq P_i \geq \frac{\epsilon}{3}F_i$, so the competitive ratio is $\frac{F_i}{\text{OPT}} \leq \frac{3}{\epsilon}$, a contradiction. □

The previous lemma shows that for most of the time steps in $[r_i, c_i]$ FIFO has m processors busy working. In the next lemma, we show that the work done by FIFO during $[r_i, c_i]$ is concentrated on jobs which did not arrive before $r_i - F_i$. We define ***processor idling steps*** to be the aggregate number of time steps per processor where the processor is not working on any job. Hence, during one time step that not all m processors are busy working, there can be at most m processor idling steps in total.

LEMMA 3.3. *During $[r_i, c_i]$, FIFO does more than $m(1+\frac{\epsilon}{3})F_i$ work on jobs which arrived after $r_i - F_i$.*

PROOF. Since J_i is the job with the maximum flow time F_i, all previous jobs must have had less flow time than F_i. Therefore, all jobs which received any processing during $[r_i, c_i]$ must have arrived at earliest $r_i - F_i$.

Now to complete the lemma we calculate the total work done during $[r_i, c_i]$. From Lemma 3.2 the number of processor idling steps is at most $m\frac{\epsilon}{3}F_i$ during $[r_i, c_i]$. Since the processors have speed $1+\epsilon$, the total work that is done during $[r_i, c_i]$ is at least

$$m(1+\epsilon)F_i - m\frac{\epsilon}{3}F_i > m(1+\frac{\epsilon}{3})F_i$$

which completes the lemma. □

Using the previous lemmas we can complete the proof.

Proof of Theorem 3.1 We consider the work of the optimal schedule. OPT achieves a flow time of $\text{OPT} < \frac{\epsilon F_i}{3}$ from the assumption that FIFO does not achieve a competitive ratio of $\frac{3}{\epsilon}$.

Consider all the jobs which arrived during $[r_i - F_i, r_i]$, OPT must finish every such job before $r_i + \frac{\epsilon}{3}F_i$. During the interval $[r_i - F_i, r_i + \frac{\epsilon}{3}F_i]$ the optimal schedule can do at most $m(1+\frac{\epsilon}{3})F_i$ work with 1 speed.

However from Lemma 3.3 the jobs which arrive after $r_i - F_i$ have more than $m(1+\frac{\epsilon}{3})F_i$ work. Hence the optimal schedule cannot possibly finish all jobs by time $r_i + \frac{\epsilon}{3}F_i$, a contradiction. □

4. UNWEIGHTED MAXIMUM FLOW TIME USING WORK STEALING

In this section, we consider a variation of work stealing, called steal-k-first work stealing scheduler, the formal definition of which will be discussed later. Our goal is to show the following theorem.

THEOREM 4.1. *The maximum unweighted flow time of the steal-k-first work stealing scheduler with* $(k+1+(k+2)\epsilon)$ *speed is* $O(\frac{1}{\epsilon^2}\max\{\text{OPT}, \ln n\})$ *for any* $k \geq 0$ *and any* $0 < \epsilon < \frac{1}{k+2}$ *with high probability.*

By scaling the constant ϵ using the constant k in Theorem 4.1, we can trivially get the Corollary below.

COROLLARY 4.2. *The maximum unweighted flow time of the steal-k-first work stealing scheduler with* $(k+1+\epsilon)$ *speed is* $O(\frac{1}{\epsilon^2}\max\{\text{OPT}, \ln n\})$ *for any* $k \geq 0$ *and any* $0 < \epsilon < 1$ *with high probability.*

For a version of steal-k-first, namely admit-first, where the constant $k = 0$, we have the following result.

COROLLARY 4.3. *The maximum unweighted flow time of the admit-first work stealing scheduler with* $(1+\epsilon)$ *speed is* $O(\frac{1}{\epsilon^2}\max\{\text{OPT}, \ln n\})$ *for any* $0 < \epsilon < 1$ *with high probability. In particular, if* $\text{OPT} \geq \ln n$*, then the scheduler is* $(1+\epsilon)$*-speed* $O(\frac{1}{\epsilon^2})$*-competitive with high probability.*

Work Stealing for a Single Job. The work stealing scheduler [5] is a distributed scheduler for scheduling a single parallel program. It dispatches work dynamically, rather than statically. Scheduling is done in a distributed manner, which results in scalability and lower overhead. Specifically, the runtime system creates a worker thread for every available core. Each worker maintains a local double-ended queue, called *deque*. When a worker generates new work (enables a ready node from the job's DAG), it pushes the work onto the bottom of its deque. When a worker finishes its current node, it pops a ready node from the bottom of its deque. If the local deque is empty, the worker becomes a *thief* and randomly picks a *victim* worker and tries to steal work from the top of the victim's deque. We assume that it takes a unit time step to steal work between workers.

Note that most of the time, workers work off their own queues and don't need to communicate with each other. Hence, this randomized work-stealing strategy is very effective in practice and the amount of scheduling and synchronization overhead is small. Therefore, work stealing is the default strategy used for executing parallel DAGs in many parallel runtime systems such as Cilk Plus, TBB, X10, and PPL [5, 19, 25, 28, 7].

Theoretically, however, because of this randomized and distributed characteristic, work stealing is not a strictly greedy strategy. However, work stealing provides strong probabilistic guarantees of linear speedup for a single job. Researchers have shown that work-stealing is provably efficient with high-probability when scheduling a single job [5].

Work Stealing for Multiple Jobs. Though work stealing scheduler is designed for scheduling a single job, we can extend it to scheduling multiple jobs in a straightforward way. In addition to the deque of each worker, a global FIFO queue is dedicated for the arrival and admission of new jobs. When a new job is released, it is inserted into the tail of the global queue. A worker will *admit* a job by popping if from the head of the global queue in a FIFO order.

Under different admission strategies, workers could choose to steal work or admit a job in different manners. In this paper, we consider a strategy, namely ***steal-k-first work stealing***, in which each worker always tries to randomly steal first and only tries to admit a new job if there are k consecutive unsuccessful steal attempts for some constant $k \geq 0$. Now we analyze the theoretical performance of steal-k-first and we present its empirical performance in Section 6.

Intuitions for Proving Theorem 4.1. As discussed in Section 2, to prove steal-k-first is competitive for maximum flow time, we need to show that it does not fall far behind the optimal schedule. We assume for the sake of contradiction that it does at some time t. Then we go back in time to a point t' where the algorithm was not far behind the optimal solution. This time is carefully defined by recursively going back in time ensuring (1) that the algorithm is always doing a significant amount of work during $[t', t]$ and (2) that we can actually find t' while ensuring (1) is true. After finding such a time t', we are able to show that while the algorithm may fall far behind the optimal schedule during $[t', t]$ due to not taking advantage of the parallelizability of jobs, it eventually is able to do a large amount of work. With faster speed, it catches up and this allows us to bound its performance. Before formally proving the theorem, we first show that steal-k-first does not idle much when there are jobs to execute.

Idling Steps in Steal-k-First. We define ***processor idling steps*** to be the aggregate number of time steps per processor where the processor is not working on a job (and is stealing instead). WLOG, we assume that each steal attempt takes 1 time step. To bound the idling time in steal-k-first's schedule, we first state a theorem from [5], which provides the bound on the time that a work stealing scheduler spends on stealing during the execution of a *single* job.

LEMMA 4.4. *During the time interval $[e_i, c_i]$ where e_i and c_i are the execution start time and completion time of a job J_i respectively, the expected number of steal attempts is bounded by $32mP_i$ where P_i is the critical-path length and m is the number of processors. Moreover, for any $\delta > 0$, the number of steal attempts is bounded by $64mP_i + 16\ln(1/\delta)$ with probability at least $1 - \delta$.*

Although the Lemma above only applies to the case of a single job, by extending it we can obtain a useful lemma for the case with n jobs. In the following lemma, let e_i denote the time that job J_i is admitted from the global queue by a processor. This is the first time the job is started.

LEMMA 4.5. *For a time interval that lies between the start time e_i and completion time c_i of a job J_i, with probability at least $1 - \frac{1}{n}$, the number of processor idling steps is bounded by $64mP_i + 32\ln(n) \leq 64m\mathrm{OPT} + 32\ln(n)$.*

PROOF. Consider Lemma 4.4 and choose $\delta = \frac{1}{n^2}$. The probability of any job J_i exceeding the idling time bound $64mP_i + 16\ln(n^2) = 64mP_i + 32\ln(n)$ during $[e_i, c_i]$ is $\frac{1}{n^2}$. This idling time bound holds for any time interval that is between $[e_i, c_i]$. Union bounding over all n jobs and subtracting from 1 yields the probability in the lemma. □

W will use the following lemma to later bound the idling time due to steal attempts between the arrival time r_i and the start time e_i of a job J_i.

LEMMA 4.6. *Under steal-k-first with a speed of $s = k+1+(k+2)\epsilon$, the number of idling steps during a time interval $[t', t]$ that is contained in $[r_i, e_i]$, the time between when a job arrives and is removed from the global queue, is at most $\frac{k}{k+1}(k+1+(k+2)\epsilon)m(t-t') + km$.*

PROOF. Every time a processor has more than k steal attempts, the processor will do one unit of work. Thus for any time interval of length $(t - t')$ there can be at most a $s\frac{k}{k+1}(t-t') + k$ steal attempts per processor. The lemma follows by aggregating over all processors. □

Now we can bound the amount of work steal-k-first does. We say that a job J_i ***spans*** a time interval $[t_a, t_{a-1}]$, if its release time $r_i \leq t_a$ and its completion time $c_i \geq t_{a-1}$.

LEMMA 4.7. *If a job spans a time interval $[t_a, t_{a-1}]$, then steal-k-first work stealing with speed $k+1+(k+2)\epsilon$ does at least $\frac{k+1+(k+2)\epsilon}{k+1}m(t_b - t_a) - (km + 64m\mathrm{OPT} + 32\ln(n))$ work with probability at least $1 - \frac{1}{n}$.*

PROOF. By definition, $[t_a, t_{a-1}]$ lies between $[r_i, c_i]$. From Lemma 4.6, the number of idling steps during $[t_a, e_i]$ is at most $\frac{k}{k+1}(k+1+(k+2)\epsilon)m(e_i - t_a) + km \leq \frac{k}{k+1}(k+1+(k+2)\epsilon)m(t_{a-1} - t_a) + km$. From Lemma 4.5, the number of idling steps during $[e_i, t_b]$ is at most $64m\mathrm{OPT} + 32\ln(n)$ with probability at least $1 - \frac{1}{n}$.

Thus, during $[t_a, t_{a-1}]$ the amount that work steal-k-first with speed $k+1+(k+2)\epsilon$ does is at least

$$\begin{aligned}
&(k+1+(k+2)\epsilon)\, m(t_{a-1} - t_a) - (64m\mathrm{OPT} + 32\ln(n)) \\
&- \left(\frac{k}{k+1}\,(k+1+(k+2)\epsilon)\, m(t_{a-1} - t_a) + km\right) \\
=&\frac{k+1+(k+2)\epsilon}{k+1} m(t_b - t_a) - (km + 64m\mathrm{OPT} + 32\ln(n))
\end{aligned}$$

with probability at least $1 - \frac{1}{n}$. □

Time Intervals in Steal-k-First. The main challenge in analyzing steal-k-first is that it is difficult to show that the remaining processing time of jobs in its queue is comparable to that of OPT's queue. Rather than directly bounding the differences between the two queues as done in previous section, we will construct a set of time intervals where steal-k-first must be busy most of the time. Using the assumption that steal-k-first has resource augmentation, we will draw a contradiction by showing that steal-k-first has completed a large amount of work which is even more than the total amount of work available during a time interval.

From here on, our goal is to show that the steal-k-first with $(k+1+(k+2)\epsilon)$-speed achieves a maximum flow time of $O(\frac{1}{\epsilon^2}\max\{\mathrm{OPT}, \ln(n)\})$ with high probability. To simplify the proof, we rewrite the objective to eliminate the max and show instead that steal-k-first achieves a maximum flow of $\frac{65}{\epsilon^2}(\mathrm{OPT} + \ln(n) + k)$, $k \geq 0$ is a constant and $0 < \epsilon < \frac{1}{k+2}$.

Let J_i be the job in steal-k-first's schedule with the maximum flow time F_i. Recall that r_i and c_i are the arrival and completion time of J_i, respectively. To show contradiction, we assume that $F_i \geq \frac{65}{\epsilon^2}(\mathrm{OPT} + \ln(n) + k)$.

We will recursively define a set of time intervals

$$T = \{[t', t_\beta], [t_\beta, t_{\beta-1}], [t_{\beta-1}, t_{\beta-2}] \ldots [t_1, t_0], [t_0, r_i], [r_i, c_i]\}$$

where $t' \leq t_\beta \leq t_{\beta-1} \leq \ldots \leq t_1 \leq t_0 \leq r_i \leq c_i$. To illustrate the time intervals, Figure 1 shows an example execution trace of steal-k-first.

Let t_0 be the arrival time of the earliest arriving job among the jobs that are not finished by steal-k-first right before time r_i. For instance, in Figure 1 there are two jobs (job J_0 and job J_q) that are active right before r_i. Among then, job J_0 has the earliest arrival time, so t_0 is defined using it. If there are no jobs right before r_i, let $t_0 = r_i$. Now we define further intervals recursively. Given the time t_{a-1}, we want to define t_a. If $t_{a-1} - t_a \leq \epsilon F_i$, then we are done defining intervals; otherwise, we define t_a as the arrival time of the earliest arriving job among those that are not finished by steal-k-first right before time t_{a-1}. We say that a certain job J_a ***defines*** an interval $[t_a, t_{a-1}]$, if it is the earliest arriving job unsatisfied by steal-k-first right before t_{a-1} and t_a is its arrival time.

Note that this process of defining intervals will always terminate. The procedure terminates when $t_{a-1} - t_a \leq \epsilon F_i$, which must happen if one goes back to the first time a job arrives. We let β denote the maximum value that a takes during this inductive definition. Hence, $[t_\beta, t_{\beta-1}]$ is the earliest time interval defined in this scheme. Moreover, the arrival time t' of the earliest arriving job among those that are unfinished right before time t_β satisfies $t' - t_\beta \leq \epsilon F_i$. As in Figure 1, interval $[t', t_\beta]$ is the first such interval that has length less than ϵF_i.

Work Done by Steal-k-First. We intend to show that steal-k-first does a lot of work during the interval $[t_\beta, c_i]$. In fact, we will show that if the assumption of $F_i \geq \frac{65}{\epsilon^2}(\mathrm{OPT} + \ln(n) + k)$ is true, then steal-k-first would have done more work than the total work of all jobs that are active during $[t_\beta, c_i]$, which is not possible and leads to a contradiction.

To do so, we partition $[t_\beta, c_i]$ into two sets of time intervals, specifically $S_1 = \{[t_a, t_{a-1}], \forall\ 0 < a \leq \beta\} \cup \{[t_0, r_i]\}$ during $[t_\beta, r_i]$, and $S_2 = \{[r_i, c_i]\}$. We first show that for intervals in S_1, steal-k-first does more work than OPT.

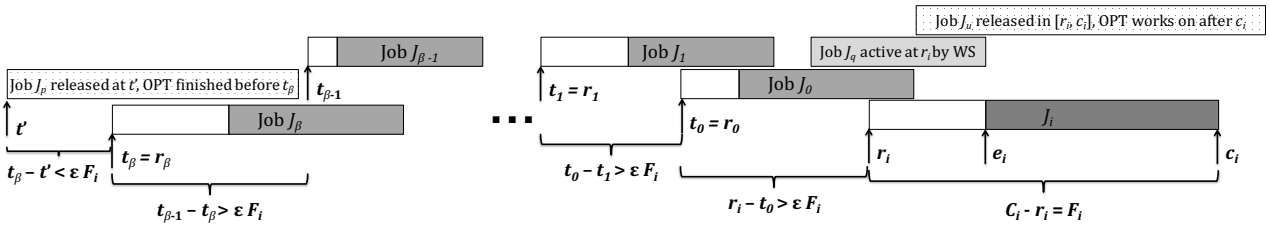


Figure 1: An example execution trace of work-stealing identifying jobs' release and completion times.

LEMMA 4.8. *For any time interval $[t_a, t_{a-1}] \in S_1$ during $[t_\beta, r_i]$, with probability at least $1 - \frac{1}{n}$ the work that steal-k-first does is more than $m(t_{a-1} - t_a)$, which is as much as* OPT *does.*

PROOF. By definition, there is a job J_a which defines this time interval. Specifically, this job spans the time interval. According to Lemma 4.7, we know that with probability $1 - \frac{1}{n}$ the amount of work steal-k-first does is at least $\frac{k+1+(k+2)\epsilon}{k+1} m(t_{a-1} - t_a) - (km + 64m\text{OPT} + 32\ln(n))$.

Recall that by assumption that $F_i > \frac{65}{\epsilon^2}(\text{OPT} + \ln(n) + k)$ and by definition that $(t_{a-1} - t_a) > \epsilon F_i$, we have

$$\begin{aligned}(t_{a-1} - t_a) &> \epsilon F_i > \frac{65}{\epsilon}(\text{OPT} + \ln(n) + k)\\ &= \frac{1}{\epsilon}\frac{1}{m}(65km + 65m\text{OPT} + 65m\ln(n))\\ &> \frac{1}{\epsilon}\frac{1}{m}(km + 64m\text{OPT} + 32\ln(n))\end{aligned}$$

Hence, $(km + 64m\text{OPT} + 32\ln(n)) < \epsilon m(t_{a-1} - t_a)$

Thus during any time interval $[t_a, t_{a-1}]$ in S_1, the work done by steal-k-first (with speed $k + 1 + (k+2)\epsilon$) on jobs is at least:

$$\begin{aligned}&\frac{k+1+(k+2)\epsilon}{k+1} m(t_{a-1} - t_a) - (km + 64m\text{OPT} + 32\ln(n))\\ &> m(t_{a-1} - t_a) + \frac{(k+2)\epsilon}{k+1} m(t_{a-1} - t_a) - \epsilon m(t_{a-1} - t_a)\\ &= m(t_{a-1} - t_a) + \frac{\epsilon}{k+1} m(t_{a-1} - t_a)\end{aligned}$$

Clearly OPT with only 1 speed can only do at most $m(t_{a-1} - t_a)$ work during this time interval. □

We now show that for S_2, steal-k-first does a lot of work too.

LEMMA 4.9. *During $[r_i, c_i] \in S_2$, the amount of work that steal-k-first does on jobs is more than $mF_i + \epsilon mF_i + m\text{OPT}$ with probability $1 - \frac{1}{n}$.*

PROOF. Consider the work that steal-k-first does during $[r_i, c_i]$. By definition this interval has a length of F_i and we know that J_i spans this interval. Directly applying Lemma 4.7, we derive that with probability $1 - \frac{1}{n}$ the amount of work done by steal-k-first during $[r_i, c_i]$ is at least

$$\begin{aligned}&\frac{k+1+(k+2)\epsilon}{k+1} mF_i - (km + 64m\text{OPT} + 32\ln(n))\\ =&mF_i + \epsilon mF_i + \frac{\epsilon}{k+1} mF_i - (km + 64m\text{OPT} + 32\ln(n))\end{aligned}$$

By definition, $0 < \epsilon < \frac{1}{k+2}$, so $\frac{1}{k+1}\frac{1}{\epsilon} > 1$. Also recall that by assumption that $F_i > \frac{65}{\epsilon^2}(\text{OPT} + \ln(n) + k)$, we have

$$\begin{aligned}\frac{\epsilon}{k+1} mF_i &> \frac{m}{k+1}\frac{65}{\epsilon}(\text{OPT} + \ln(n) + k)\\ &> 65m(\text{OPT} + \ln(n) + k)\\ &> (km + 64m\text{OPT} + 32\ln(n)) + m\text{OPT}\end{aligned}$$

Hence, $\frac{\epsilon}{k+1} mF_i - (km + 64m\text{OPT} + 32\ln(n)) > m\text{OPT}$. Therefore, the amount of work done by steal-k-first during $[r_i, c_i]$ is more than $mF_i + \epsilon mF_i + m\text{OPT}$ with probability $1 - \frac{1}{n}$. □

We need one more critical argument to complete the analysis. The reason we defined these time intervals inductively is to identify the jobs that are active under steal-k-first during $[t_\beta, c_i]$. The total volume of these jobs is bounded by the work that OPT can finish. However, just showing that steal-k-first does more work than OPT during $[t_\beta, c_i]$ will not suffice, as OPT could have done part of this work either before t_β or after c_i. As shown in Figure 1, the two jobs (job J_p and job J_u) in dotted shade are executed by steal-k-first during $[t_\beta, c_i]$, while OPT finished job J_p before t_β and started working on job J_u after c_i. The next lemma bounds the maximum amount of work that are available for steal-k-first to work on during $[t_\beta, c_i]$.

LEMMA 4.10. *For jobs that are active under steal-k-first during $[t_\beta, c_i]$, their total amount of work is at most $m(r_i - t_\beta) + m(\epsilon F_i + \text{OPT} + F_i)$.*

PROOF. By definition, $[t_\beta, c_i]$ consists of time intervals of S_1 during $[t_\beta, r_i]$ and time interval of $S_2 = \{[r_i, c_i]\}$. Also recall that the length of interval $[r_i, c_i]$ is F_i. Hence, the total length of $[t_\beta, c_i]$ is $(r_i - t_\beta) + F_i$.

Moreover, by definition of t_β, the earliest arriving job that is unsatisfied by steal-k-first just before time t_β must have arrived no earlier than time $t_\beta - \epsilon F_i$. Thus, the jobs that are active under steal-k-first during $[t_\beta, c_i]$ all arrived during $[t_\beta - \epsilon F_i, c_i]$.

Further, all these jobs have an optimal maximum flow time no more than OPT under the optimal scheduler. Therefore, OPT must be able to complete all of them by time $c_i + \text{OPT}$. Knowing that OPT can only work on these jobs during $[t_\beta - \epsilon F_i, c_i + \text{OPT}]$, the total amount of work of those jobs can have volume at most $m(r_i - t_\beta) + m(\epsilon F_i + \text{OPT} + F_i)$. □

Finally, we are ready to complete the proof.

Proof of Theorem 4.1 To prove the theorem, we consider the jobs that are active under steal-k-first during $[t_\beta, c_i]$. By Lemma 4.10, we know that the total amount of work of these jobs, denoted as X, is bounded: $X \le m(r_i - t_\beta) + m(\epsilon F_i + \text{OPT} + F_i)$. Note that these jobs are the only ones available for steal-k-first to work on during $[t_\beta, c_i]$. Therefore, during

$[t_\beta, c_i]$ steal-k-first cannot do more than X work even with speedup.

On the other hand, consider the minimum amount of work that steal-k-first must have done during $[t_\beta, c_i]$, denoted as Y, assuming that $F_i > \frac{65}{\epsilon^2}(\mathrm{OPT} + \ln(n) + k)$ is true. We will see that $Y > X$, which leads to a contradiction.

From Lemma 4.8, we know that during $[t_\beta, r_i]$ the amount of work steal-k-first does is more than

$$m\left(\sum_{0<a\leq\beta}(t_{a-1}-t_a)+(r_i-t_0)\right) = m(r_i - t_\beta)$$

From Lemma 4.9, we know that during $[r_i, c_i]$, steal-k-first does more than $mF_i + \epsilon mF_i + m\mathrm{OPT}$ work. Thus, for interval $[t_\beta, c_i]$, we get $Y > m(r_i - t_\beta) + mF_i + \epsilon mF_i + m\mathrm{OPT}$.

Now we compare X and Y:

$$\begin{aligned} Y - X >& m(r_i - t_\beta) + mF_i + \epsilon mF_i + m\mathrm{OPT} \\ & - m(r_i - t_\beta) - m(\epsilon F_i + \mathrm{OPT} + F_i) > 0 \end{aligned}$$

Hence, $Y > X$. In other words, if the assumption of F_i is true, during $[r_i, c_i]$ steal-k-first must have done more work than the total available work, which gives a contradiction.

Thus, we obtain the theorem. □

Discussion about Steal-k-First. Note that for steal-k-first work stealing with $k = 0$, instead of steal first, this scheduler will in fact admit all jobs from the global queue first. We denote this special case as ***admit-first***. From Theorem 4.1, we know that the theoretical performance of steal-k-first is better with smaller constant k. Hence, admit-first has the best theoretical performance and is $O(\frac{1}{\epsilon^2})$-competitive with high probability with $1+\epsilon$ speed, as it guarantees that a job's execution is not delayed by unnecessary random stealing.

However, as shown in Section 6 steal-k-first for a relatively large k performs better than admit-first empirically. Intuitively, if there is any job available for stealing, then in expectation m consecutive random steal attempts would be able to find the stealable work. Thus, for $k \geq m$, steal-k-first better approximates FIFO, which we know works well.

In contrast, in admit-first jobs could run sequentially when there are more than m unfinished jobs. During these times, jobs at the end of the global queue take long time to be admitted and they further take longer time to finish sequential execution in the worst case. Hence, this could increase the maximum flow time of the system.

Moreover, steal-k-first requires a speed of more than $(k+1)$ theoretically to be competitive, mainly due to the worst case scenario where each job has a unit time of work but takes k stealing steps to admit. However, in practice, jobs have much larger work and the constant k steal attempts for admitting a job is negligible in practice.

5. WORK STEALING LOWER BOUND

In this section we give a lower bound for the work stealing algorithm. We show that in the online setting, the scheduler when given any constant speed is $\Omega(\log n)$ competitive. This shows that our upper bound analysis of the algorithm is effectively tight.

LEMMA 5.1. *Work stealing is $\Omega(\log n)$-competitive for maximum flow time in the online setting when given any constant resource augmentation.*

PROOF. Let n be an input parameter and let the number of machines be $m = \log n$. Let s be a constant specifying the resource augmentation given to work stealing. The schedule consists of n jobs, which are identical. A job consists of one task which is the predecessor of $m/10$ independent tasks. Note that the total work of the job is $m/10 + 1$ and can be competed by a 1 speed scheduler scheduler in 2 time steps. A single job is released at multiples of $2m$ starting at time 0. Note that even if a job is executed sequentially, it will complete in only $m/10 + 1$ time steps. Thus, these jobs do not have overlapping times where multiple jobs are alive in any non-idling schedule.

Now fix any job and consider the probability that the job executes completely sequentially by a work stealing scheduler. This occurs if every steal attempt fails to find the processor holding the tasks for the job. In a single time step, the probability that $m-1$ processors do not successfully steal is $(1 - \frac{1}{m-1})^{m-1} \geq \frac{1}{2e}$ for sufficiently large m. The probability that all processors fail to steal for $m/10$ time steps is greater than $(\frac{1}{2e})^{m/10}$.

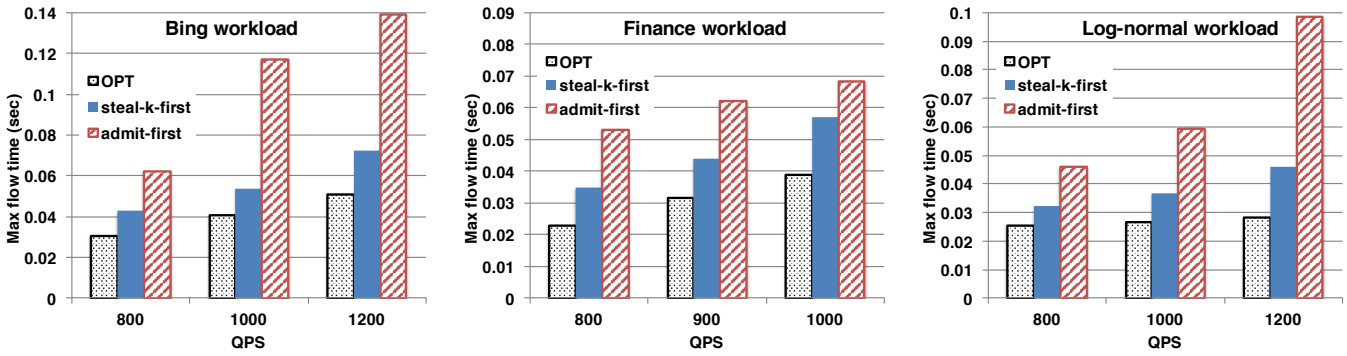
Now consider the expected number of jobs which execute sequentially by work stealing. There are $n = 2^m$ jobs released. The expected number of jobs to execute sequentially is $2^m(\frac{1}{2e})^{m/10} \geq 1$. Thus, the expected maximum flow time of work stealing with s speed is $\frac{m/10+1}{s} = \frac{\log n}{s}$. Knowing that the optimal solution has maximum flow time 2 and $s = O(1)$, the lemma follows. □

6. EXPERIMENTAL RESULTS FOR UNWEIGHTED MAXIMUM FLOW TIME

In this section we present the experimental results using realistic and synthetic workloads to compare the performance of OPT and two work stealing strategies: (1) *Admit-first* where workers preferentially admit jobs from the global queue and only steal if the queue is empty, and (2) *Steal-k-first* where workers preferentially steal and only admit a new job if k steal attempts fail (we use $k = 16$). Our experiments indicate that steal-k-first performs better and is almost comparable to an optimal scheduler.

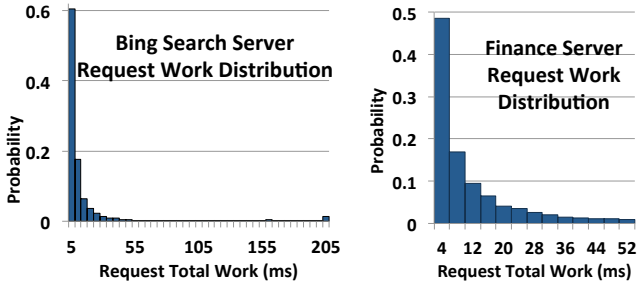
Setup: We conduct experiments on a server with dual eight-core Intel Xeon 2.4Ghz processors with 64GB main memory. The server runs Linux version 3.13.0, with processor throttling, sleeping, and hyper-threading disabled. The work-stealing algorithms are implemented in Intel Thread Building Block (TBB) [25] version 4.3, a well-engineered popular work-stealing runtime library. We extended TBB to schedule multiple jobs arriving online by adding a global FIFO queue for admitting jobs and we implement both admit-first and steal-k-first.

Since we do not know the optimal scheduler, we must approximate it using a simulated scheduler by reducing a parallel scheduling problem to a sequential scheduling problem on a single processor. In particular, for this lower bound, we assume that there is no preemption overhead and that each job can get linear speedup (fully parallelizable). Therefore, we can execute each job one at a time assuming it is a sequential job with execution time equal to its W/m where W is its total work. We then run all jobs using FIFO which is optimal in this setting. When jobs are fully parallelizable, this reduces the problem to the case where there is only one machine. In this setting, it is well known that FIFO is opti-



(a) Bing workload (b) Finance workload (c) Log-normal workload

Figure 2: Experimental results comparing the maximum flow time running on three work distributions with three different load settings and scheduled using simulated OPT, steal-k-first, and admit-first (from left to right). Note that the scale of the y-axis for the figures differ. From all different settings, OPT has the smallest max flow time, while admit-first has the largest max flow time.



(a) Bing search (b) Finance server

Figure 3: Work distribution of two workload: Bing web search [21] and an option pricing finance server [26].

mal for maximum flow time [8]. Thus, this scheduler has the performance for maximum flow time that is at least as good as any feasible scheduler, including the optimal schedule.

Workloads: We evaluate different strategies on work distributions from two real-world applications shown in Figure 3 and additional synthetic workloads with log-normal distribution. Henceforth we shall refer to workload generated from the three distributions as the *Bing workload*, the *finance workload* and the *log-normal workload*, respectively. For each distribution, we select a set of queries-per-second, QPS, to generate workloads with low ($\sim$ 50%), medium ($\sim$ 60%), and high ($\sim$ 70%) machine utilization respectively, and the inter-arrival time between jobs is generated by a Poisson process with a mean equal to $1/QPS$. Each job contains CPU-intensive computation and is parallelized using parallel for loops. 100,000 jobs are used to obtain a single point in the experiments.

Figure 2 shows the experimental results comparing simulated OPT, steal-k-first and admit-first under three different work distributions and three different load settings (i.e., query-per-second). The experiments indicate that, even though our results on OPT are lower bounds on maximum flow time, steal-k-first performs comparably to OPT — matching our intuition that it is a closer approximation for maximum flow time, as discussed at the end of Section 4.

Recall that steal-k-first has worse theoretical performance than admit-first. However, in practice, admit-first generally performs worse in terms of maximum flow time and the performance difference increases as load increases (for instance, for Bing and log-normal workloads with high utilization, admit first has twice the maximum flow). This matches our intuition — at higher loads, admit-first executes jobs more or less sequentially, while steal-first provides parallelism to already admitted jobs before admitting new jobs. Therefore steal-first is closer to FIFO in that it tries to execute jobs that arrived earlier with more parallelism. Therefore, in practice, steal-first is likely to be a good implementation for schedulers that want to minimize maximum flow time without incurring the large overheads of FIFO.

7. MAXIMUM WEIGHTED FLOW TIME

In this section our goal is to prove that the algorithm Biggest-Weight-First (BWF) is a scalable algorithm for minimizing the maximum weighted flow time.

THEOREM 7.1. *Biggest-Weight-First (BWF) is* $(1+\epsilon)$ *speed* $O(\frac{1}{\epsilon^2})$ *competitive for minimizing the maximum weighted flow for any* $\epsilon > 0$.

BWF is defined as follows, which is similar to FIFO except that priority is given to the jobs with the biggest weight. At any time t, BWF orders the jobs in decreasing order by their weight, breaking ties arbitrarily. The algorithm then assigns all of the ready nodes for the first job to some processor, then recursively does the same for the next job in the list. This continues until all processors have been assigned some node or there are no more ready nodes available. Like FIFO, BWF may have a choice on which ready nodes of a job to schedule if the job has more ready nodes than the number of processors which have not been assigned to a node when the job is considered. In this case, we assume the scheduler chooses an arbitrary set of ready nodes.

The reminder of this section is devoted to proving Theorem 7.1. For the rest of this section, we assume that BWF is given $(1+3\epsilon)$-speed for some constant $0 < \epsilon < \frac{1}{3}$ and we will show that BWF is $\frac{3}{\epsilon^2}$ competitive. Fix any sequence of jobs and let OPT denote the optimal schedule on this instance as well as the optimal maximum weighted flow time. Let F_a^* be the flow time of a job J_a in OPT.

Let J_i be the job in BWF's schedule with the maximum weighted flow time $w_i F_i$. For the sake of contradiction, we

assume that $w_i F_i > \frac{3}{\epsilon^2}\text{OPT}$. Since $\text{OPT} = w_i F_i^*$, $F_i > \frac{3}{\epsilon^2}F_i^*$, where F_i^* is the flow time of J_i in OPT. By comparing the weight w_i of job J_i, any jobs with weight at least w_i are referred as ***heavy*** jobs, and any jobs with less weight than w_i are referred as ***light*** jobs.

Time Intervals in BWF. Similar to the time intervals specified in Section 4, we will inductively define a set of time intervals

$$T = \{[t', t_\beta], [t_\beta, t_{\beta-1}], [t_{\beta-1}, t_{\beta-2}] \dots [t_1, t_0], [t_0, r_i], [r_i, c_i]\}$$

where $t' \le t_\beta \le t_{\beta-1} \le \dots \le t_1 \le t_0 \le r_i \le c_i$.

Recall that r_i and c_i are the arrival and completion time of J_i, respectively. Consider the *heavy* jobs that BWF is scheduling right before r_i. Let t_0 be the arrival time of the *earliest* arriving one of those jobs. If there are no heavy jobs right before r_i, let $t_0 = r_i$. Now we define further intervals recursively. Given the times t_{a-1}, we want to define t_a. If $t_{a-1} - t_a \le \epsilon F_i$, then we are done defining time intervals; otherwise, we define t_a to be the arrival time of the earliest arriving heavy job J_a that are unsatisfied under BWF right before time t_{a-1}. Again if there are no heavy jobs unsatisfied by BWF just before time t_{a-1} then let $t_a = t_{a-1}$. We let β denote the maximum value that a takes during this inductive definition. Hence, $[t_\beta, t_{\beta-1}]$ is the earliest time interval defined in this scheme.

Note that this process of defining intervals is almost the same as in Section 4. The only difference is that the job J_a, which defines the interval $[t_a, t_{a-1}]$, is the earliest unfinished *heavy* job under *BWF*. We only consider heavy jobs, because under BWF only heavy jobs can preempt job J_i and other heavy jobs and any light jobs can only execute when all the available nodes of all the active heavy jobs are already executing by some processors. Thus, when analyzing the flow time of J_i and other heavy jobs, we can ignore the remaining light jobs, since light jobs cannot in interfere the execution of heavy ones. Hence, the processor idling steps in the remaining of this section is refering to the time steps where a processor is not working on nodes corresponding to heavy jobs.

We begin the proof by showing that during all time intervals between $[t_\beta, r_i]$, BWF is using most time steps to process nodes for heavy jobs.

LEMMA 7.2. *During any interval $[t_a, t_{a-1}]$ where $a \le k$, the number of processor idling steps (where a processor is not working on nodes corresponding to heavy jobs) is at most $m\frac{\epsilon^2}{3}F_i$.*

PROOF. For the sake of contradiction, assume that this is not true. Then consider the job that defines $[t_a, t_{a-1}]$ and let this job be J_a. By definition this heavy job arrived at t_a and is still being processed at time t_{a-1}. From BWF's scheduling policy, every time step during $[t_a, t_{a-1}]$, where some processors find no nodes from heavy jobs to work on, all ready nodes of J_a are being scheduled. Hence the processors are decreasing the remaining critical path of J_a at these times by Proposition 2.1. Since the job is not finished until at t_{a-1}, this job must have a critical-path length P_a longer than $P_a > t_{a-1} - t_a > \frac{\epsilon^2}{3}F_i$. Also since J_a is a heavy job and $w_a \ge w_i$ and by assumption $w_i F_i > \frac{3}{\epsilon^2}\text{OPT}$, its weighted flow time is at least

$$w_a(t_{a-1} - t_a) > w_a\frac{\epsilon^2}{3}F_i \ge w_i\frac{\epsilon^2}{3}F_i > \frac{\epsilon^2}{3}\frac{3}{\epsilon^2}\text{OPT} \ge \text{OPT}$$

However, OPT cannot complete a job faster than its critical-path length, so $F_a^* \ge P_a$. Further, J_a's weighted flow time under OPTis at most the maximum weighted flow time OPT. We have

$$\text{OPT} \ge w_a F_a^* \ge w_a P_a > w_a(t_{a-1} - t_a) > \text{OPT}$$

This gives a contradiction. □

Using the previous lemma, we bound the aggregate amount of work done by BWF on heavy jobs during $[t_\beta, r_i]$.

LEMMA 7.3. *During $[t_\beta, r_i]$, the amount of work that BWF does on heavy jobs is more than $m(1+2\epsilon)(r_i - t_\beta)$.*

PROOF. From Lemma 7.2, we know that there are only $m\frac{\epsilon^2}{3}F_i$ processor idling steps where a processor is not working on nodes corresponding to heavy jobs during any time interval $[t_a, t_{a-1}]$. In addition, we know $t_{a-1} - t_a > \epsilon F_i$, since $a \le \beta$. Hence, the work done by BWF (with $1+3\epsilon$ speed) on heavy jobs during $[t_a, t_{a-1}]$ is at least:

$$\begin{aligned} &m(1+3\epsilon)(t_{a-1} - t_a) - m\frac{\epsilon^2}{3}F_i \\ >&m(1+3\epsilon)(t_{a-1} - t_a) - m\frac{\epsilon}{3}(t_{a-1} - t_a) \\ >&m(1+2\epsilon)(t_{a-1} - t_a) \end{aligned}$$

Summing over all the intervals yields the lemma. □

Similarly, we can bound the amount of work done by BWF on heavy jobs during $[r_i, c_i]$.

LEMMA 7.4. *During $[r_i, c_i]$, the amount of work that BWF does on heavy jobs is more than $m(1+2\epsilon)F_i$.*

PROOF. By assumption, $F^* < \frac{\epsilon^2}{3}F_i$. Since OPT cannot finish a job in less time than its critical-path length, job J_i has $P_i \le F_i^* < \frac{\epsilon^2}{3}F_i$. From Proposition 2.1, we can derive that the number of processor idling steps where a processor is not working on heavy jobs is at most mP_i. Hence, the amount of work done by BWF during $[r_i, c_i]$ is at least $m(1+3\epsilon)F_i - mP_i > m(1+3\epsilon)F_i - m\frac{\epsilon^2}{3}F_i > m(1+2\epsilon)F_i$, since $\epsilon < \frac{1}{3}$. □

Now we bounds the maximum amount of work that are available for BWF to work on during $[t_\beta, c_i]$.

LEMMA 7.5. *For jobs that are active under BWF during $[t_\beta, c_i]$, their total amount of work is at most $m(r_i - t_\beta) + m(1+\epsilon+\frac{\epsilon^2}{3})F_i$.*

PROOF. By definition, the total length of $[t_\beta, c_i]$ is $(r_i - t_\beta) + F_i$. Moreover, by definition of t_β, the earliest arriving heavy job that is unsatisfied BWF just before time t_β must have arrived no earlier than time $t_\beta - \epsilon F_i$. Thus, the heavy jobs that are active under BWF during $[t_\beta, c_i]$ all arrived during $[t_\beta - \epsilon F_i, c_i]$.

Furthermore, all these heavy jobs have an optimal maximum weighted flow time no more than OPT under the optimal scheduler, i.e., $\text{OPT} \ge F_a^* w_a$. By definition of a heavy job $w_a \ge w_i$ and by assumption $w_i F_i > \frac{3}{\epsilon^2}\text{OPT}$, we have $w_a F_i \ge w_i F_i > \frac{3}{\epsilon^2}\text{OPT} > \frac{3}{\epsilon^2}F_a^* w_a$. Thus, the flow time F_a^* of these heavy jobs under the optimal schedule is $F_a^* < \frac{\epsilon^2}{3}F_i$.

Therefore, OPT must be able to complete all of them by time $c_i + \frac{\epsilon^2}{3}F_i$. Knowing that OPT can only work on these

jobs during $[t_\beta - \epsilon F_i, c_i + \frac{\epsilon^2}{3}F_i]$, the total amount of work of those jobs can have volume at most $m(r_i - t_\beta + F_i) + m(\epsilon F_i + \frac{\epsilon^2}{3}F_i) = m(r_i - t_\beta) + m(1 + \epsilon + \frac{\epsilon^2}{3})F_i$. □

Finally, we are ready to complete the proof.

Proof of Theorem 7.1 To prove the theorem, we consider the heavy jobs that are active under BWF during $[t_\beta, c_i]$. By Lemma 7.5, we know that the total amount of work of these jobs, denoted as X, is bounded: $X \leq m(r_i - t_\beta) + m(1 + \epsilon + \frac{\epsilon^2}{3})F_i$. Note that these jobs are the only ones available for BWF to work on, so during $[t_\beta, c_i]$ BWF cannot do more than X work even with speedup.

On the other hand, consider the minimum amount of work that BWF must have done during $[t_\beta, c_i]$, denoted as Y, assuming that $w_i F_i > \frac{3}{\epsilon^2}\text{OPT}$ is true. We will see that $Y > X$, which leads to a contradiction.

From Lemma 7.3, we know that during $[t_\beta, r_i]$ the amount of work BWF does is more than $m(1 + 2\epsilon)(r_i - t_\beta)$ From Lemma 7.4, we know that during $[r_i, c_i]$, BWF does more than $m(1 + 2\epsilon)F_i$ work. Thus, for interval $[t_\beta, c_i]$, we get $Y > m(r_i - t_\beta) + m(1 + 2\epsilon)F_i$.

Now we compare X and Y and note that $\epsilon < \frac{1}{3}$:

$$\begin{aligned} Y - X >& m(r_i - t_\beta) + m(1 + 2\epsilon)F_i \\ & - m(r_i - t_\beta) - m(1 + \epsilon + \frac{\epsilon^2}{3})F_i > 0 \end{aligned}$$

Hence, if the assumption of $w_i F_i$ is true, then during $[r_i, c_i]$ BWF must have done more work than the total available work, which gives a contradiction. By scaling ϵ, we obtain the theorem. □

Remarks. The result of weighted flow time can be applied to maximum stretch. In the sequential setting, weighted flow time captures maximum stretch by setting the weight to be the inverse of the processing time. In other words, the flow of a job is scaled by the inverse of its processing time in the stretch objective for sequential jobs. However, stretch is not well-defined for DAG jobs. In particular, should the flow time be scaled by the inverse of the total work or the critical path length? Although there are two natural interpretations of the stretch in the DAG setting, both of them can be still captured by weighted flow time. Since BWF is $(1 + \epsilon)$-speed $O(\frac{1}{\epsilon^2})$-competitive for maximum weighted flow time and there are strong lower bounds without speed augmentation, so this result can be viewed as essentially the best positive theoretical result for maximum stretch.

8. RELATED WORK

In scheduling theory, two dominant models have emerged for modeling the parallelizability of jobs. One model is the Directed-Acyclic-Graph (DAG) model, which was considered in this paper. This model is a good model for representing parallel programs written using programming languages and libraries designed for this purpose, and, due to this the model is well connected to practice [5, 14, 13, 12, 6].

The other model considered is known as the arbitrary speed-up curves setting when online and sometimes referred to as the malleable task setting when offline. In the arbitrary speed-up curves setting, each job J_j consists of μ_j phases and the ith phase is associated with a tuple $(p_{i,j}, \Gamma_{i,j}(m'))$. The value of $p_{i,j}$ is the work of the ith phase for job j and $\Gamma_{i,j}(m')$ is a speed-up function that specifies the rate $p_{i,j}$ is processed at when job J_j is given m' processors when in the ith phase. The phases of the job must be processed sequentially and $\Gamma_{i,j}$ specifies the parallelizability of J_i during phase i. It is generally assumed that $\Gamma_{i,j}$ is a non-decreasing sublinear function.

Both models have been widely considered both online and offline. However, it is important to note that the models are fundamentally different and there does not appear to be a straightforward way of translating results from one model to the other. In particular, in the DAG setting, the realized parallelizability of a job, at any point in time, depends on the nodes of the job that are free to execute. That is, the nodes whose predecessors have been previously processed. The realized parallelizability, or number of ready nodes, depends not only on how much work on the job has been complete in the past, but also *which* nodes were processed. Due to this, one cannot map an arbitrary DAG to a set of speed-up curves since the parallelizability of a job in the speed-up curves model only depends on the amount of work previously processed. Alternatively, in the speed-up curves model, it could be the case that a job has a speed up function of $\Gamma(m') = \sqrt{m'}$. In this case, a job is processed at a rate of $\sqrt{m'}$ when given m' processors for any $1 \leq m' \leq m$. In the DAG setting, one cannot simulate this speed-up curves since the parallelizability of a job is essentially linear up to the number of nodes ready to be scheduled.

The results mentioned in Section 1 on minimizing the maximum (weighted) flow time were all for the case where jobs are sequential. When jobs are parallelizable, few online algorithmic techniques are known to have strong performance guarantees for maximum flow time, unlike for other objectives such as average flow time [11]. The only positive result on maximum flow time is for the arbitrary speed-up curves setting, where a $(1 + \epsilon)$-speed $O(\log n)$-competitive algorithm for unweighted flow time was shown [24]. This result is complemented by a lower bound showing that no algorithm can be s-speed $o(\log n)$-competitive for any constant resource augmentation $s > 0$. The lower bound is somewhat surprising because average flow time, typically thought of as a harder objective than maximum flow time, admits an $O(1)$-competitive algorithm with constant resource augmentation [11]. Thus, this result on maximum flow time seems to close the problem and rules out hope for a $O(1)$-competitive algorithm even with resource augmentation. Our result, which provides $O(1)$-competitive algorithm with $1 + \epsilon$ speed augmentation is the first to cleanly separate the DAG model from the arbitrary speed-up curves model for any objective.

Work on the speed-up curves setting has also considered other objectives. A $(1+\epsilon)$ speed $O(\frac{1}{\epsilon})$-competitive algorithm for any $\epsilon > 0$ is known for average flow time [11]. Without resource augmentation, an $O(\log P)$-competitive algorithm is known assuming polynomial speed-up functions under some assumptions of the jobs parallelizability [18]. For the ℓ_k-norms of flow time, a $(1 + \epsilon)$ speed $O(\frac{k}{\epsilon^{2k+1}})$ for any $\epsilon > 0$ is known [10, 15]. Notice that maximum flow time is captured by the ℓ_k-norms for sufficiently large k, but here the competitive ratio grows with k. For maximum flow time a $(1+\epsilon)$-speed $O(\log n)$-competitive is known with a matching lower bound on any algorithm given $O(1)$ speed augmentation [24].

For the DAG scheduling setting, little is known online. Recently some works have considered algorithms in the real time scheduling setting [27, 22].

9. CONCLUSION

The DAG model has been influential in design of theoretically good and practically efficient schedulers for executing single parallel program. In this paper, we give the first results in this model for maximum flow time, an important scheduling metric, for multiprogrammed environment where jobs arrive online. This work opens up many interesting further questions. For instance, is resource augmentation absolutely necessary for DAG jobs? Also, are there online algorithms with strong performance guarantees for other objectives such as the ℓ_k-norms of flow time? Our results also show that the online scheduling of parallel programs in the DAG model differs from the arbitrary speed-up curves setting. It would be of interest to further explore connections and differences between these two models.

Acknowledgment

This research was supported in part by a Google Research Award, a Yahoo Research Award, NSF awards CCF-1337218, CCF-1218017, and CCF-1150036.

10. REFERENCES

[1] Kunal Agrawal, Charles E Leiserson, Yuxiong He, and Wen Jing Hsu. Adaptive work-stealing with parallelism feedback. *ACM Trans. Computer Syst.*, 26(3):7, 2008.

[2] Kunal Agrawal, Jing Li, Kefu Lu, and Benjamin Moseley. Scheduling parallel DAG jobs online to minimize average flow time. In *SODA '16*, pages 176–189, 2016.

[3] Christoph Ambühl and Monaldo Mastrolilli. On-line scheduling to minimize max flow time: an optimal preemptive algorithm. *Oper. Res. Lett.*, 33(6):597–602, 2005.

[4] Michael A. Bender, Soumen Chakrabarti, and S. Muthukrishnan. Flow and stretch metrics for scheduling continuous job streams. In *SODA '98*, pages 270–279, 1998.

[5] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *JACM*, 46(5):720–748, 1999.

[6] F. Warren Burton and M. Ronan Sleep. Executing functional programs on a virtual tree of processors. In *FPCA*, pages 187–194. ACM, 1981.

[7] Colin Campbell and Ade Miller. *A Parallel Programming with Microsoft Visual C++: Design Patterns for Decomposition and Coordination on Multicore Architectures*. Microsoft Press, 2011.

[8] Chandra Chekuri, Sungjin Im, and Benjamin Moseley. Online scheduling to minimize maximum response time and maximum delay factor. *Theory of Computing*, 8(1):165–195, 2012.

[9] Thomas H Cormen, Charles E Leiserson, and Ronald L Rivest. *Introduction to algorithms*. MIT press, 2009.

[10] Jeff Edmonds, Sungjin Im, and Benjamin Moseley. Online scalable scheduling for the ℓ_k-norms of flow time without conservation of work. In *SODA '11*, pages 109–119, 2011.

[11] Jeff Edmonds and Kirk Pruhs. Scalably scheduling processes with arbitrary speedup curves. *ACM Trans. Algo.*, 8(3):28, 2012.

[12] Rainer Feldmann, Peter Mysliwietz, and Burkhard Monien. Studying overheads in massively parallel min/max-tree evaluation. In *SPAA '94*, pages 94–103, 1994.

[13] Vincent W. Freeh, David K. Lowenthal, and Gregory R. Andrews. Distributed Filaments: Efficient fine-grain parallelism on a cluster of workstations. In *OSDI '94*, pages 201–213. USENIX, 1994.

[14] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *PLDI*, pages 212–223, 1998.

[15] Anupam Gupta, Sungjin Im, Ravishankar Krishnaswamy, Benjamin Moseley, and Kirk Pruhs. Scheduling jobs with varying parallelizability to reduce variance. In *SPAA '10*, pages 11–20, 2010.

[16] Yuxiong He, Wen-Jing Hsu, and Charles E Leiserson. Provably efficient online nonclairvoyant adaptive scheduling. *IEEE Trans. Parallel Distrib. Syst.*, 19(9):1263–1279, 2008.

[17] Sungjin Im, Benjamin Moseley, and Kirk Pruhs. A tutorial on amortized local competitiveness in online scheduling. *SIGACT News*, 42(2):83–97, 2011.

[18] Sungjin Im, Benjamin Moseley, Kirk Pruhs, and Eric Torng. Competitively scheduling tasks with intermediate parallelizability. In *SPAA '14*, pages 22–29, 2014.

[19] Intel. Intel CilkPlus, Sep 2013. https://www.cilkplus.org/.

[20] Bala Kalyanasundaram and Kirk Pruhs. Speed is as powerful as clairvoyance. *J. ACM*, 47(4):617–643, 2000.

[21] Saehoon Kim, Yuxiong He, Seung-won Hwang, Sameh Elnikety, and Seungjin Choi. Delayed-dynamic-selective (DDS) prediction for reducing extreme tail latency in web search. In *WSDM '15*, pages 7–16, 2015.

[22] Jing Li, Jian-Jia Chen, Kunal Agrawal, Chenyang Lu, Christopher D. Gill, and Abusayeed Saifullah. Analysis of federated and global scheduling for parallel real-time tasks. In *ECRTS '14*, pages 85–96, 2014.

[23] OpenMP. OpenMP Application Program Interface v4.0, July 2013. http://http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf.

[24] Kirk Pruhs, Julien Robert, and Nicolas Schabanel. Minimizing maximum flowtime of jobs with arbitrary parallelizability. In *WAOA '10*, pages 237–248, 2010.

[25] James Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. O'Reilly Media, 2010.

[26] Shaolei Ren, Yuxiong He, Sameh Elnikety, and Kathryn S McKinley. Exploiting processor heterogeneity in interactive services. In *ICAC '13*, pages 45–58, 2013.

[27] Abusayeed Saifullah, David Ferry, Jing Li, Kunal Agrawal, Chenyang Lu, and Christopher D. Gill. Parallel real-time scheduling of dags. *IEEE Trans. Parallel Distrib. Syst.*, 25(12):3242–3252, 2014.

[28] Olivier Tardieu, Haichuan Wang, and Haibo Lin. A work-stealing scheduler for x10's task parallelism with suspension. In *PPoPP '12*, 2012.